

AD-A193 681

PROGRAMMING N-CUBES WITH A GRAPHICAL PARALLEL
PROGRAMMING ENVIRONMENT VER. (U) WASHINGTON UNIV
SEATTLE DEPT OF COMPUTER SCIENCE K GATES ET AL. NOV 86

1/1

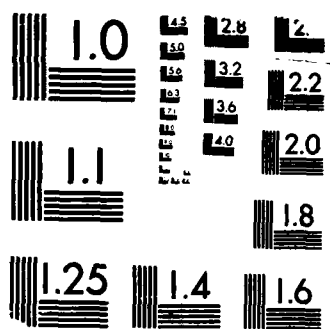
UNCLASSIFIED

TR-86-12-02 N00014-85-K-0328

F/G 12/5

NL





MICROCOPY RESOLUTION TEST CHART
 NATIONAL BUREAU OF STANDARDS-1963-A

DTIC FILE COPY

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

4

AD-A193 681

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER none	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) Programming N-cubes with a Graphical Parallel Programming Environment vs. an Extended Sequential Language		5. TYPE OF REPORT & PERIOD COVERED Technical Report
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Kevin Gates and David Socha		8. CONTRACT OR GRANT NUMBER(s) N00014-85-K-0328
9. PERFORMING ORGANIZATION NAME AND ADDRESS University of Washington Department of Computer Science Seattle, Washington 98195		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Office of Naval Research Information Systems Program Arlington, VA 22227		12. REPORT DATE November 1986
		13. NUMBER OF PAGES 10
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)		15. SECURITY CLASS. (of this report) Unclassified
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE
16. DISTRIBUTION STATEMENT (of this Report) Distribution of this report is unlimited.		
17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)		
18. SUPPLEMENTARY NOTES		
19. KEY WORDS (Continue on reverse side if necessary and identify by block number) programming languages, parallel programming, N-cube architectures, Cholesky algorithm, programming environment		
20. ABSTRACT (Continue on reverse side if necessary and identify by block number) We compare the writing and execution of programs written in Cosmic Cube C with programs written in the graphical parallel programming environment and language Poker. Our example programs, an implementation of a Cholesky algorithm for a banded matrix, were written in both languages and compiled into object codes that ran on the Cosmic Cube. However the program written in Poker is shorter, faster and easier to write, easier to debug and portable without changes to other parallel architectures. The Poker program was slower than the program written directly in Cosmic Cube C, however the experiments provided		

DTIC
ELECTE
APR 13 1988
S
OE

Disturbs into changes that make Poker programs nearly as fast.

DD 1 JAN 73 1473

EDITION OF 1 NOV 65 IS OBSOLETE

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

Programming N-cubes With a Graphical
Parallel Programming Environment
Versus an Extended Sequential Language


Kevin Gates
Department of Applied Math, FS-20

David Socha
Department of Computer Science, FR-35

University of Washington

Technical Report 86-12-02

November 1986

Distribution For	
DTIC	<input checked="" type="checkbox"/>
Prescribed	<input type="checkbox"/>
Unprescribed	<input type="checkbox"/>
Classification	
By	
Distribution/	
Availability Codes	
Avail and/or	Special
A-1	
	

Abstract

We compare the writing and execution of programs written in Cosmic Cube C with programs written in the graphical parallel programming environment and language Poker. Our example programs, an implementation of a Cholesky algorithm for a banded matrix, were written in both languages and compiled into object codes that ran on the Cosmic Cube. However the program written in Poker is shorter, faster and easier to write, easier to debug, and portable without changes to other parallel architectures. The Poker program was slower than the program written directly in Cosmic Cube C, however the experiments provided insights into changes that make Poker programs nearly as fast.

88 4 12 070

Programming N-cubes With a Graphical Parallel Programming Environment *Versus* an Extended Sequential Language [†]

Kevin Gates
Department of Applied Mathematics, FS-20

David Socha
Department of Computer Science, FR-35
University of Washington

Abstract

We compare the writing and execution of programs written in Cosmic Cube C with programs written in the graphical parallel programming environment and language Poker. Our example programs, an implementation of a Cholesky algorithm for a banded matrix, were written in both languages and compiled into object codes that ran on the Cosmic Cube. However the program written in Poker is shorter, faster and easier to write, easier to debug, and portable without changes to other parallel architectures. The Poker program was slower than the program written directly in Cosmic Cube C, however the experiments provided insights into changes that make Poker programs nearly as fast.

1 Introduction

Now that there are a number of parallel architectures, including n-cubes, there is an increased need for better parallel programming techniques. In particular, programmers need assistance handling the parallel aspects of their programs. One direction for improvement is in the design of better parallel programming languages and environments. At one extreme are very high level languages such as Crystal [1] in which the programmer relies on the compiler to extract parallelism and optimize inter-process communication. At the other extreme are low level languages/language-support-systems such as crystalline [2] which provide bare-bones message facilities and no algorithmic programming assistance but are tuned to maximizing execution speed. Poker [3] takes a more conservative intermediate position providing a high-level parallel programming abstraction supported by a graphical programming environment and language while still relying upon the programmer to extract the parallelism from algorithms. The key question is whether such a high-level language can provide this programming support while not compromising the speed of the programs.

This paper addresses this question by discussing the programming and execution of the modified Cholesky decomposition written in both Poker and the extended C [4] for the Cosmic Cube [5]. We provide timing results for the execution of both programs on the Cosmic Cube.

2 The Algorithmic Test Case

Our basis of comparison between the two programming languages is an algorithm to solve the matrix equation:

[†] Supported in part by National Science Foundation Grant DCR-8416878 and by Office of Naval Research Contract No. N00014-85-K-0328.

$$A * x = b$$

where A is a symmetric, positive definite, diagonally dominant matrix of size $n \times n$, with semi-bandwidth β . The solution method that we use is an implementation of the modified Cholesky algorithm:

$$A = L * D * L^T$$

where L is a lower triangular matrix with semi-bandwidth β , and D is a diagonal matrix. The complete solution of the matrix problem is then found through the solution of the following three related equations:

$$L * z = b \tag{1}$$

$$D * y = z \tag{2}$$

$$L^T * x = y \tag{3}$$

We solve equation 1 with forward substitution, equation 2 by a vector division, and equation 3 with back substitution.

The modified Cholesky algorithm has seven steps:

1. Load matrix A .
2. Load vector b .
3. Do Cholesky decomposition $A = L * D * L^T$.
4. Do forward substitution $L * z = b$.
5. Calculate $D * y = z$.
6. Do backward substitution $L^T * x = y$.
7. Dump result vector x .

The algorithm uses $(\beta + 1)^2$ processes conceptually arranged in a square grid. Steps 3, 4, and 6 requires that each process be able to broadcast to every other process in its column, and that each process on the diagonal be able to broadcast across its row. Instead of fully connecting the columns and rows we acknowledged the high cost of message passing in current implementations of n-cubes and did our own message routing, using hypercube connections for the columns and trees for the rows. These communication needs neatly fit into a hypercube interconnection scheme.

3 The Languages

Both Cosmic Cube C and Poker use a non-shared memory model of cooperating sequential processes communicating by message passing and a modified C to express the sequential details of the process's code. They differ chiefly in how they handle the parallel aspects of a program. The Cosmic Cube C provides no parallel structure while Poker uses a graphical environment for describing the parallel aspects of the program.

3.1 The Poker Language and Environment

Poker is both a parallel programming language for non-shared memory algorithms, and a parallel programming environment used to program and execute Poker programs. The Poker environment runs on a conventional computer. It allows the user to write Poker programs and serves as the front end when running Poker programs on the simulator, emulator, or target parallel architecture. Both the Poker language and the Poker environment are non-standard and intimately connected, so this section will interleave discussion of their features from the point of view of a programmer.

The description of a Poker program is based on the graph model so often used to describe a parallel program. This description has five logical components:

```

for i ← 0 to n - 1 do (i = update row)
  for j ← i to min(n, i + β) do (j = update column)
    for k ← max(0, i - β) to i do (k = pivot row)
      if k = i then
         $d_{kk} = a_{kk}$ 
         $a_{kj} \leftarrow a_{kj} / d_{kk}$ 
      else
         $a_{ij} \leftarrow a_{ij} - (a_{ik} * a_{kj})$ 
      endif
    endfor
  endfor
endfor.

```

Figure 1: The Cholesky Decomposition Algorithm.

1. a graph whose vertices are processes and whose edges are communication channels between the processes,
2. a labeling of the vertices with the codes to run in the processes,
3. a labeling of the edges at each vertex,
4. the codes for the processes, and
5. a description of the inputs and outputs (the Cholesky decomposition has none).

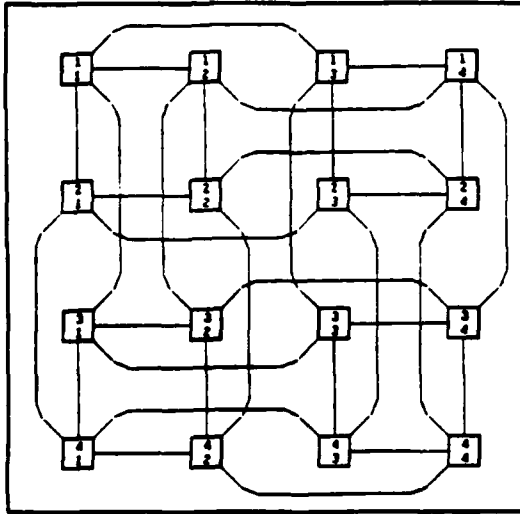
Poker encodes parallel algorithms at this same level of abstraction, using separate "views" of a graphical programming environment to define four of the five parts; the process codes are programmed using a standard text editor.¹ For example, a Poker programmer for the Cholesky Decomposition step (step 3), Figure 1, codes the five components as shown in Figure 2:

1. *Communication Graph*: The programmer uses a mouse or number pad to explicitly *draw* the connections between the processes (boxes). This drawing is the *only* definition of the interconnection; there is no textual specification of the interconnection.
2. *Process Assignment*: The programmer assigns the code to run in each process, by entering the code name at the top of the process box. Note that the Communication Graph is still visible here, aiding the programming of the Process Assignment.
3. *Port Name Assignment*: The programmer labels the edges from the perspective of each vertex; each edge has two names, one for each end or "port." Again, the communication graph is visible to aid the programmer.
4. *Process Definition*: The (usually small number of) process codes are written in a slightly modified version of C [6]. Section 3.2 on Poker C describes the modifications. Figure 2 shows only the code for pivot.
5. *Stream Name Assignment*: The dangling edges of the graph connect the algorithm to its inputs and outputs. The programmer labels dangling edges with the name of the input and output streams.²

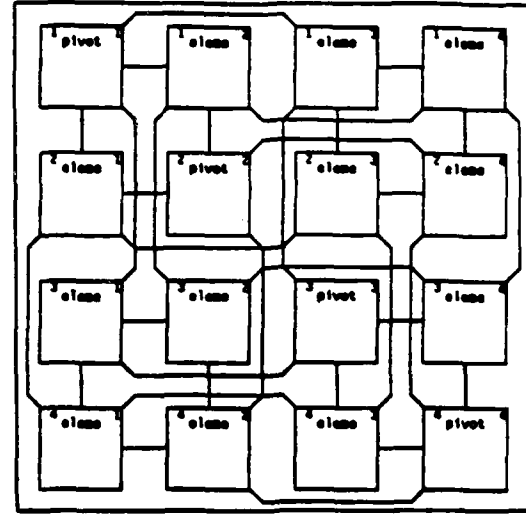
¹ The term "View" refers to the manner in which Poker programs are stored and recalled. The five parts form a database defining the program. Each "View" is a perspective on this database, perhaps incorporating information from more than one of the parts of the program, as in the graph programmed in the Communication Graph View is visible in the Process Assignment View.

² A stream is a series of data values of arbitrary, and possibly mixed, types. Poker requires that the stream data in files are basic data types, e.g. bool, char, short, and so on.

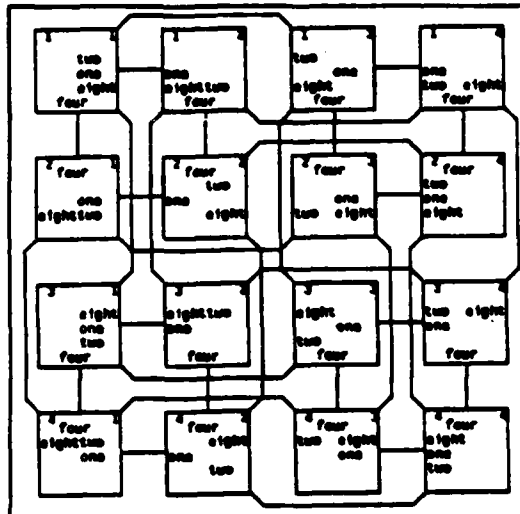
Communication Graph



Process Assignment



Port Name Assignment



Process Definition

```
code pivot;
ports one,two,four,eight;
pivot() {
  short MatrixRow, MatrixCol;
  float A[2][4], b[4];
  port Rec[2], Column[2], SendPort, RecvPort;

  import MatrixRow from MatrixRow;
  import MatrixCol from MatrixCol;
  import A from A;
  Rec[1] = one;

  for (PivotRow = 1; PivotRow < 16; PivotRow += 1) {
    if (PivotRow == MatrixRow + 4*UpdateRow) {
      All[0] = A[0][UpdateRow];
      Recv(All, MatrixRow, Recv);

      SendPort = Column[PivotRow % 2];
      SendPort <- piv;
    }
  }
}
```

Stream Name Assignment

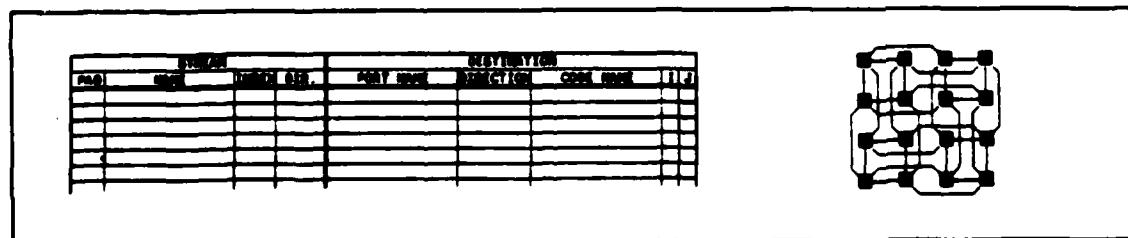


Figure 2: The five parts of a Poker program for the Cholesky decomposition step.

These pictures are exact replicas of the lower part of the corresponding Poker views, showing the parallel parts of the Poker program as entered by the user. The upper part of a Poker View contains additional status information including the number of processes in the program (between 4 and 4096 processes) and the state of the programming environment during programming. Note that several views display the communication graph as context while programming the remaining parts of the parallel algorithm.

Three additional Views aid the user in initializing the size of the program and compiling and executing it:

- *CHiP Parameters View*: Used to select the logical CHiP architecture³, including the number of processes.
- *Command Request View*: Used to compile, load, link, and so on, Poker programs so that they are ready to execute.
- *Trace View*: Used to watch the execution of the algorithm. The Trace View uses the same picture as in the Process Assignment, except that the last four lines of each box show the current values of up to four variables being traced in that process.

The Trace View is especially important when debugging Poker programs. The Poker generic simulator runs on a conventional von-Neuman computer and provides a much more interactive environment in which to debug Poker programs before tying up the Cosmic Cube or other parallel computer. If additional problems arise during Cosmic Cube execution, we can turn the tracing mechanism on to try to pinpoint the error, though such tracing changes the timing of the program making it exceedingly difficult to pinpoint timing problems.

What we have seen so far is one "phase" of a Poker program; that is, one unit of parallel execution. Most parallel programs are more complex, requiring a number of different parallel units, which we call phases. Each of these phases is defined using the five parts described above and a grid of processes of the same size. Processes occupying the same location on the grid in different phases may pass information between phases, using the *import/export* convention described in Section 3.2. Each phase may connect the processes in any graph independent of the connections used in other phases.

Each phase executes as a unit and the processes synchronize after completing each phase. Phases can be invoked manually, from the Poker environment, or via a program written in a sequential phase-control language.

3.2 Poker C

Poker C code is standard C [6] with a few changes.⁴ Using the Process Definition in Figure 2 as an example, we see that each code has a header specifying the name of the code (*pivot*), and the ports (*one*, *two*, *four*, *eight*) followed by a set of C routines. Variables exist only within routines; there are no external variables.

Processes occupying the same location on the grid communicate through an inter-phase data space that they, and only they, share. Variables in this inter-phase data space do not exist before being assigned values. However, once assigned, the variable exists forever. Hence, the most recently assigned value is available until the program terminates. The expression

`export(local, inter-phase)`

stores the value of the locally defined variable *local* into the inter-phase variable *inter-phase*. At any point in the future, from any phase, the expression

`import(local, inter-phase)`

³Poker was designed to program the CHiP architecture. Both CHiP Parameters View and the switches in the Communication Graph are vestiges of this decision.

⁴A more complete description of Poker C is found in [7].

loads the local variable *local* with the last value stored into *inter-phase*.

The statements

```
port <- expression
variable <- port
```

send and receive messages. *Port* is either a port name from the header's *ports* list, or a variable of type *port*. The run-time system checks receive messages to make sure that the message received is of the same type as the *variable*; incompatible types cause a fatal run-time error. Messages between processes may be any of the basic data types, arrays with statically defined size, or structures, as long as there are no pointers in the data.

4 Implementations

Since both languages use the same non-shared memory paradigm of communicating sequential processes and the same base language, C, the process codes are very similar in the two languages. However, we had to do some subtle programming in the Cosmic Cube C program to determine which messages to keep and which to pass. In our message passing scheme communications are difficult in the Cosmic Cube C version, but painless in Poker since Poker automatically encodes the "direction" of a message by the port from which it enters.

Both programs have three types of processes: (1) a host "controller" process that allocates a cube, spawns processes onto the Cosmic Cube, and controls their execution, (2) a host "file server" process that passes values between files and processes, and (3) 16 cube processes, one per node of a 4-cube, implementing the Cholesky algorithm. The Cosmic Cube C version combined the controller and file server into one process.

5 Mapping the Guest Graph to the Host Graph

Both programs used the communication links of an n-cube. We mapped the n-cube graph of the algorithm (guest graph) directly onto the n-cube of the host architecture (host graph) maintaining the node adjacencies. In general the mapping of the algorithm's interconnection, the guest graph, to the host graph of the architecture can be more complex.

Cosmic Cube C provides no explicit concept of either graph; instead the cube is logically completely connected and the definition and use of the guest graph is embedded into the program. This increases the difficulty of changing the mapping of the guest graph to the host graph. This difficulty could be eliminated by writing a more general spawning/message-passing scheme.

Poker simplifies the mapping problem by separating it from the program specification. The guest graph, defined by the Communication Graph, is automatically mapped onto the host graph and the resulting mapping stored in a file. The spawner reads this file at run-time to determine the process placement on the processors. Changing the mapping file changes the resultant mapping with no need to recompile.

This is an obvious place for an automatic mapping system. Currently, Poker does one mapping, a row-major assignment of process/node numbers to processes. It is easy to imagine a more sophisticated system that would try to improve the mapping, perhaps first using simulation to weight the use of the communication paths, and also automatically mapping from a family of guest graphs to a family of host graphs. Berman [8] describes one such system. This is a rich area for further research.

6 Results

The cube processes ran on an otherwise unloaded 4-cube, to avoid message contention with other cube programs, and the host processes ran on a Sun 2/120 directly connected to the cube, to avoid ethernet

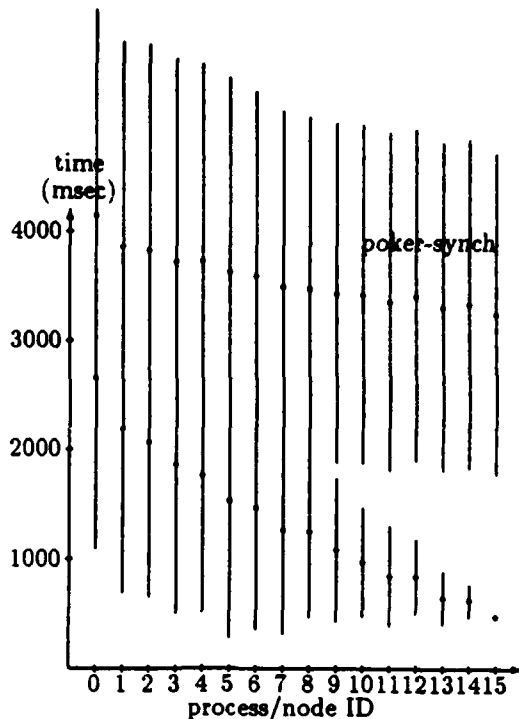


Figure 3: Timings for Poker program with synchronization between phases.

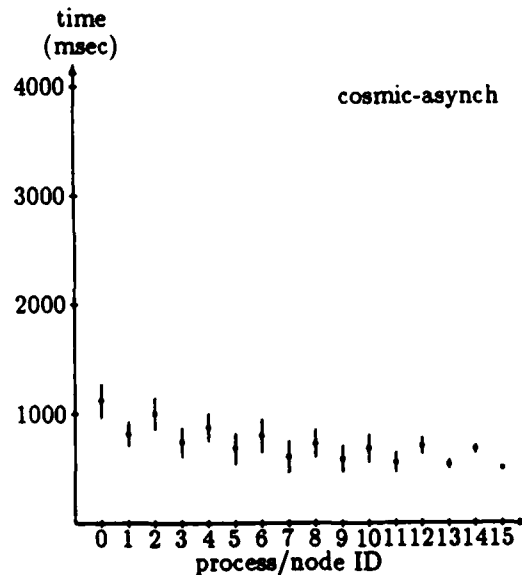


Figure 4: Timings for Cosmic Cube C program with no synchronization between steps.

delays. We tried to keep the Sun otherwise unloaded during the runs. Each cube process cached its timing values until the end of the program to minimize measurement perturbations.

To estimate the variance we ran each program 8 or more times.⁵ Each cube process recorded the start and end time of steps 3, 4-5, and 6 (step 5 was folded into step 4). This provided two measurements for each process: the "total" time from start of step 3 to end of step 6, and the "combined" time, a sum of the times between the beginning and end of steps 3-6.

The process/node number is on the x-axis (recall that there is one process/node) while the y-axis shows the execution time in milliseconds. The vertical bars are the 95% confidence regions for the averages.

Figure 3 emphasizes two trends. First, the cost of synchronization, estimated by the difference between the "combined" times (lower points) and "total" times, is quite large, ranging from 40% to 80% of the total execution time.

Second, processes with higher numbers have faster execution times, especially for the combined times. This reduction has at least two causes: (1) the controller starts the cube processes in ascending order of process number so that lower numbered processes block on input from higher numbered processes, (2) the lower numbered processes need to forward messages between the host, connected to node 0, and the other processes. The Cholesky algorithm, with its low computation/communication ratio, accentuates this high communication cost. Also note that higher numbered processes have much less variance for the "combined" times indicating that message forwarding or idling for messages from processes that have yet to execute takes

⁵ We attempted to execute 15 runs of each of the four program discussed below, randomly shuffling the executions into a sequence of 60 runs. However the Cosmic Cube has been having problems recently, so we were unable to get one set of 60 runs. The number of runs per figure are: 12 for Fig. 6, 13 for Fig. 6, and 8 for Fig. 5. The 15 runs for Fig. 4 came from an earlier batch. Note how the variance decreases with increasing number of runs.

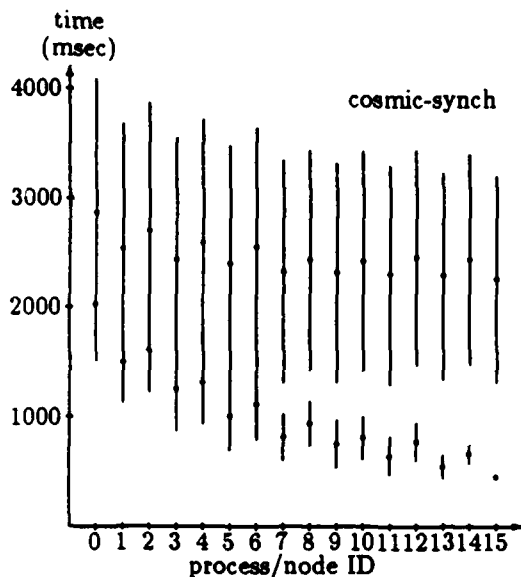


Figure 5: Timings for Cosmic Cube C program with synchronization between steps.

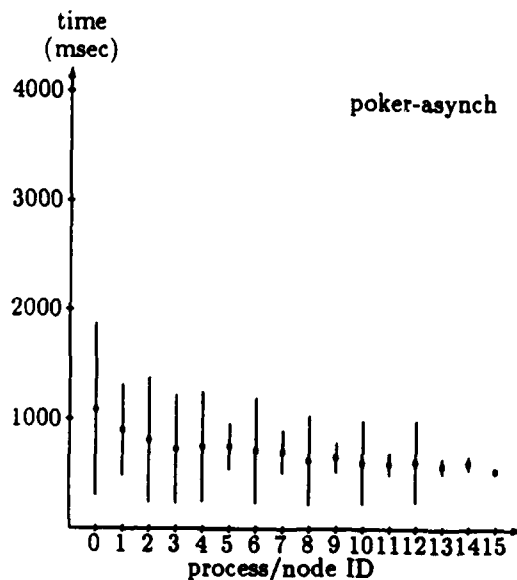


Figure 6: Timings for Poker program without synchronization between phases.

a heavy toll on lower numbered processes.

For comparison, Figure 4 shows the execution times for the Cosmic Cube C program. There is no synchronization between steps of this program so the total time equals the combined time. This program runs about 3 times faster than the Poker program. However the difference between "combined" Poker times and the Cosmic Cube C times decreased as the node number increases, suggesting that inter-step synchronization is killing Poker programs. Figures 5 and 6 demonstrate that the Poker program is not inherently slower than the Cosmic Cube C program. Figure 5 shows that the times for the same Cosmic Cube C program, but with synchronization between steps, approach those of synchronized Poker. One reason the synchronized Cosmic Cube C program does not slow down as much as the synchronized Poker program is that the controller and file server for the Cosmic Cube C program are one process, so the Cosmic Cube C program has about one third fewer synchronization messages than the Poker program. Going in the other direction, Figure 6 shows that the same Poker program without the inter-phase synchronization is as fast as the Cosmic Cube C program.

In conclusion, The Poker Cholesky program exhibits the same run-times as the Cosmic Cube C program, indicating that Poker's higher level programming abstraction does not sacrifice an efficient implementation for at least one communication intensive algorithm.

Programming in raw Cosmic Cube C could still be advantageous since it allows the following: message passing to simulate a completely connected graph of processes, dynamic process creation and deletion, dynamic reallocation of the cube, and the use of the programmer's knowledge of a program to optimize its run time with non-blocking sends and receives and no synchronization.

On the other hand, Poker provides the following: a higher level program abstraction that eases the definition of a parallel program, a visually oriented environment, automatic provision of routines to handle file input/output, process spawning, cube allocation, and process control so that the programmer can concentrate on the algorithm, and a parallel simulator/debugger for developing programs off-line. In addition, programs written in Poker are portable to different parallel architectures [9].

7 Improving the efficiency of Cosmic Poker

These experiments pointed to places where more effort on the part of compilers, host operating systems, or host hardware might improve the efficiency of parallel programs. The amount of improvement depends on the algorithm, its implementation, and the n-cube hardware available. In particular, Poker programs potentially could benefit from:

- Running phases without intervening synchronization. As a result of this work, we have decided that Cosmic Poker will provide a way to run phases asynchronously.
- Broadcasting, fanning-out, or fanning-in messages between the controller and cube processes. Support in the host operating system or hardware could substantially reduce synchronization costs. Fan-in synchronization could be made extremely cheap with the use of a hardware "AND" line raised when all processes have completed some task, such as a phase.
- Placing the controller on the cube, say on the opposite side from the host, to balance the message load and avoid the host/cube bottleneck.
- Using non-blocking sends and receives where possible. This requires extensive flow analysis.
- Statically initializing the message descriptors.

In most cases there are tradeoffs so that the modifications would speed some types of algorithms on some machines, and slow others.

8 Conclusion

This and other experiments indicate that high-level parallel programming languages and environments can substantially elevate the programmability of n-cubes while still producing efficient code. This is not to say that parallel programs will always be easier to write and execute more efficiently if written in Poker. An algorithm that *requires* complete connectivity or dynamic creation or destruction of processes does not fit into the current Poker framework. However, Poker does provide a cleaner approach to writing efficient programs for a large class of parallel algorithms.

9 Acknowledgments

A number of people assisted us in this project: Our advisors Larry Snyder and Loyce Adams provided motivation, advise, and helpful criticism; Bob Mitchell wrote portions of the Poker C compiler; James Schaad helped retarget Poker to the Cosmic Cube, producing good code under time pressure; Beth Ong helped with the design and programming of the numerical algorithm; Chuck Seitz and his group at Caltech were most helpful in providing access to the Cosmic Cube and its hosts and dealing promptly with our problems; Phil Nelson, Mary Bailey, and many fellow graduate students made helpful comments throughout the project. We thank all of these people.

References

- [1] Marina C. Chen. Very high-level parallel programming in crystal. *Proceedings of the Second Conference on Hypercube Multiprocessors*, 1986.
- [2] Brian Beckman. Hypercube operating systems: development for performance and programmability. *Proceedings of the Second Conference on Hypercube Multiprocessors*, 1986.

- [3] Lawrence Snyder. Parallel programming and the Poker programming environment. *Computer*, 17(7):27-36, July 1984.
- [4] Wen-King Su, Reese Faucette, and Chuck Seitz. *C Programmer's Guide to the Cosmic Cube*. Technical Report 5203:TR:85, Computer Science Department, California Institute of Technology, September 1985.
- [5] Charles L. Seitz. The cosmic cube. *Communications of the ACM*, 28(1):22-33, January 1985.
- [6] Brian W. Kernighan and Dennis M. Richie. *The C Programming Language*. Academic Press, New York, 1978.
- [7] Lawrence Snyder. *Poker (4.0) Programmer's Reference Guide*. Technical Report 86-05-04, Computer Science Department, University of Washington, November 1986.
- [8] Francine Berman, Michael Goodrich, Charles Koelbel, III W. J. Robison, and Karen Showell. Prep-P: a mapping preprocessor for CHiP architectures. *Proceedings of the 1985 International Conference on Parallel Processing*, 731-733, 1985.
- [9] Lawrence Snyder and David Socha. Poker on the cosmic cube: the first retargettable parallel programming language and environment. *Proceedings of the International Conference on Parallel Processing, IEEE*, 628-635, 1986.

END

DATE

FILMED

7-88

Dtic